BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 3276

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT No. 5

1 January 1976 to 31 March 1976

Principal Investigator: Mr. Frank E. Heart
Telephone (617) 491-1850, Ext. 470

Sponsored by:
Advanced Research Projects Agency
ARPA Order No. 2351, Amendment 15
Program Element Codes 62301E, 62706E, 62708E

Contract No. F08606-75-C-0032
Effective Date: 1 January 1975
Expiration Date: 19 July 1976
Contract Amount: $2,533,832

Title of Work: Operation and Maintenance of the ARPANET

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Advanced Research Projects Agency or the U.S. Government.

Report No. 3276

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT No. 5

1 January 1976 to 31 March 1976

Principal Investigator: Mr. Frank E. Heart
Telephone (617) 491-1850, Ext. 470

Sponsored by:
Advanced Research Projects Agency
ARPA Order No. 2351, Amendment 15
Program Element Codes 62301E, 62706E, 62708E

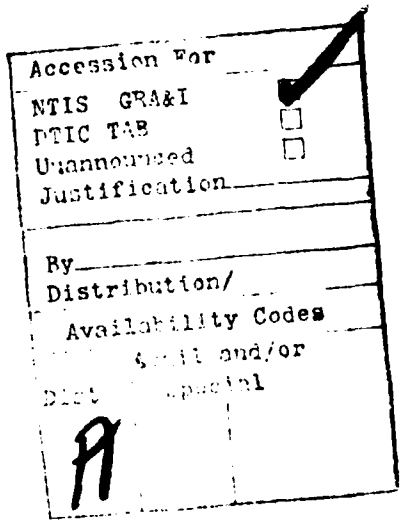
Contract No. F08606-75-C-0032
Effective Date: 1 January 1975
Expiration Date: 19 July 1976
Contract Amount: $2,533,832

Title of Work: Operation and Maintenance of the ARPANET

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

TABLE OF CONTENTS



1. INTRODUCTION

This Quarterly Technical Report, Number 5, describes aspects of our work performed under Contract No. F08606-75-C-0032 during the first quarter of 1976. The previous reports in this series dealt largely with work quite closely related to the development, maintenance, and operation of the ARPANET, e.g., the IMPs and TIPs of the ARPANET and the Satellite IMPs and PLIs connected to the ARPANET. However, beginning with this quarter, our work with the ARPANET has been largely funded under a contract from the Defense Communications Agency and our work with Satellite IMPs, PLIs, etc. has been supported under new contracts from ARPA which will be reported elsewhere. The only significant body of work still funded under this contract is a study into the feasibility of using the Pluribus computer as the basis of a large, secure message-switching system. The remainder of this document describes our Pluribus message switch work.

2. PLURIBUS MESSAGE SWITCH STUDY

The message handling systems which have been developed and used within the ARPANET have spurred widespread interest. We are studying extensions of this technology, contemplating a system which will provide a message handling service of large capacity and high reliability while meeting stringent security requirements. Our task is to explore the suitability of the Pluribus computer for this application. This interim report on our work will be followed by a comprehensive final report to appear in our next Quarterly Technical Report.

Message switching, not inherently a difficult task, becomes difficult when a large number of messages must be processed in an environment requiring high reliability and complex and stringent rules regarding who should and should not have access to each message. A secure system requires means for guaranteeing not only that messages are not misdirected to improper recipients but further that there is no method by which an individual can obtain access to messages without authorization. To guarantee the integrity of both the operation and the security of a message switching system requires a highly reliable system with good fault tolerance. Thus for a large message switch of the kind contemplated, the hardware and operating system together must:

- Be able to support a high volume of traffic, with provision for expansion.
- Be highly reliable and highly available.
- Assure against release of information to unauthorized recipients.

2.1 Application of the Pluribus to Message Switching

The Pluribus multiprocessor, developed at BBN under ARPA support for the ARPANET, has many characteristics which make it attractive as a machine for the message switch. In this study, we are considering how it handles, or can be made to handle, the various problems presented by this application. This will, of course, involve special software, although a substantial part of the reliability/availability software which already exists may be used. Also, some special hardware for helping with the security problems will likely be required. However, it seems possible to capitalize on the multi-resource nature of the Pluribus in coping with these problems and thereby to minimize the amount of new hardware work required.

The structure of the Pluribus has been described in previous Quarterly Technical Reports, and is further described in References 1 to 5. With the structure of the Pluribus in mind, we turn to consideration of how the Pluribus meets the needs of a large message switch.

The ability to handle a high volume of messages is a very important requirement of the message switch. A prototype installation which is being used as the basis for estimates requires on the order of 15,000 messages to be processed daily for about 2000 users. We estimate that the present TENEX system operated in a dedicated fashion could handle perhaps 60 such users. This is not surprising since the TENEX system was designed to provide efficient service of a very different type. The Pluribus, on the other hand, seems quite well suited to providing the required processing bandwidth. It was designed to be able to process, in an economical fashion, large numbers

of small tasks which could be executed in parallel. Just as in packet switching, the task of message processing has a large element of inherent and obvious parallelism stemming from the independence of the individual messages. The Pluribus was designed to gain speed by taking advantage of such parallelism.

Another characteristic of many jobs, and message processing is no exception, is that the capacities required inevitably change with time -- usually by increasing. The modularity of the Pluribus hardware structure, plus the approach we have taken in using the processors as general purpose workers, plus a highly adaptive approach in locating and utilizing available hardware resources, combine to create a system which can be adapted easily to changing requirements. Handling increased traffic should require only the addition of the needed extra hardware resources. This would typically mean adding more communication line controllers in the I/O, perhaps more memory for buffering, and perhaps more processor busses to increase processing bandwidth. Such changes do not require changes in the software, because the adaptive mechanisms necessary to incorporate shifting hardware resources are built into the software at the outset for reasons of reliability. The system thus can grow with a minimum of effort. We have set the upper bounds on growth (as determined by address fields, etc.) very high, so that large amounts of I/O and processors can be accommodated.

The need for reliability hardly needs emphasis. If all messages flow through a central message switch, when it breaks, the flow of messages stops. The approach we have taken with the Pluribus attempts to guarantee that service will rarely be interrupted and that when it is, the system will recover automatically and quickly. Service should be resumed within seconds

with any failed component excised from the system by the program. This will result in a system in which single errors have extremely low probability of stopping operations for more than a few seconds.

The third major requirement is multi-level security, the need to provide protection for several levels of access. Many different levels of messages will be handled by the same system and yet unauthorized access must be prevented. This is a difficult job. A "security kernel" in the software is one traditional approach to be considered. However, just as in the case of reliability, the multi-resource aspect of the Pluribus offers some conceptually simple approaches to the security problem.

The focus of attention in security studies has traditionally been on software and on providing assurance that no amount of ingenuity could circumvent the protective features provided in the program. However, to our knowledge, there has been little work on attempting to cope with the ways in which hardware failures can jeopardize security. The assumption of unfailing hardware is unrealistic in the practical world; any prudent system design must allow for the effects of a failure. In the Pluribus, powerful techniques have been developed for detecting and dealing with the effects of failures, both software and hardware, on the integrity of system operation. A good deal of the program (although only a small fraction of processing bandwidth) is given over to mechanisms for isolating and recovering from failures. Methods have been developed whereby the processors work together to certify functioning and to eliminate bad parts of the system so that they do not damage overall operation. Many of these same approaches can be used to assure the integrity of

system security as well as system operation. This is a vital issue and the Pluribus appears to offer an environment in which security might be insulated from the damaging effects of failures -- perhaps both hardware and software.

2.2 Security

The message switch is intended to serve a military community which has multi-level security requirements. This stringent requirement has occupied much of our attention, and we feel that we have significant progress to report.

The message switch design must insure that neither a single hardware error nor a software bug can compromise security. Unauthorized access to messages resulting from hardware errors can be prevented by performing all operations twice on different hardware, accepting the result only if both agree. This brute force approach is unnecessarily expensive, since we have found a way to organize the system so that only a small part of the code need be executed twice. Software problems can be avoided by verification of all of the code; this too is impractical, because of the large size of the system. We have devised a system organization and special hardware that permit us to verify only a small part of the code, again without compromising security.

An important idea in both of these simplifications is checksumming messages. A message complete with checksum can be moved through the system without danger even though it is handled by unverified code executing only once, providing that we insure before delivery that the checksum is still valid. The important fact is that the list of addressees is included in the checksummed data, so any alteration of either the message or an addressee will be detected by an incorrect checksum. For example, if any hardware or software failure alters the message by changing an addressee or including someone else's data, the checksum will be incorrect and the message will not be delivered.

We thus have three significant results to report:

- Checksumming messages permits us to leave much of the code unprotected, since failures in it cannot compromise security. This means that only part of the program need be executed twice, and that only part of the code need be verified.

- In the areas where double execution is required, we have clean solutions to the problems of forking a single computation into parallel computations and joining parallel computations into a single computation.

- We have devised a hardware modification for Pluribus to permit us to execute unverified code without fear that it might compromise security.

The remainder of this section presents our ideas on obtaining the requisite security in the message switch. Section 2.2.1 discusses checksumming, the concept that permits us to fragment the system into critical and non-critical parts. Briefly, a message with a checksum, referred to hereafter as a Safe Message, can be moved with impunity around the system, providing only that we insure that it is delivered only to those entitled to see it. In Section 2.2.2 we present the technique for dealing with hardware errors. For convenience in exposition we label as "orange" those non-critical parts of the code that need be executed only once, and we label the two parallel paths "green" and "blue" for those critical portions of the code in which double execution is necessary. The interesting ideas are in the fork where orange separates into green and blue, and in the join where green and blue become orange. In Section 2.2.3 we describe the hardware

that insures that errors in the unverified parts of the software cannot permit a security leak. There is a well known technique developed for a different purpose that can be used here: special hardware in the processor to support an Executive mode and a User mode. In Exec all of the computer's abilities are available, while in User mode the hardware enforces restrictions. All of the unverified code is executed in User mode. This code may include bugs, but the User mode hardware provides the needed guarantee that there can be no security leak. The idea is simple. In User mode the processor is restricted so that it can access only a limited part of the system's address space. Before the unverified code starts to run, verified code insures that there is no one else's data in that area.

2.2.1 Message Checksumming and the "Safe Message"

The basic concept that underlies our solution to the security problem is message checksumming. We define a "Safe Message" to be a fully composed message with a checksum, the checksum to include all of the addressees and the originator, as well as the text of the message. (Actually, two checksums are kept, one green and one blue.) Figure 1 shows the fields of a Safe Message. A Safe Message may move through the system without concern, providing only that there are suitable safeguards on its creation and delivery and that any legitimate alterations are handled properly.

A basic operating rule in the message switch is that a user may obtain access to only those messages for which he is an addressee. Before the user gets access to a Safe Message, a two-step check is performed. First the checksum is verified. Next, if it is correct, the system verifies that the user is included

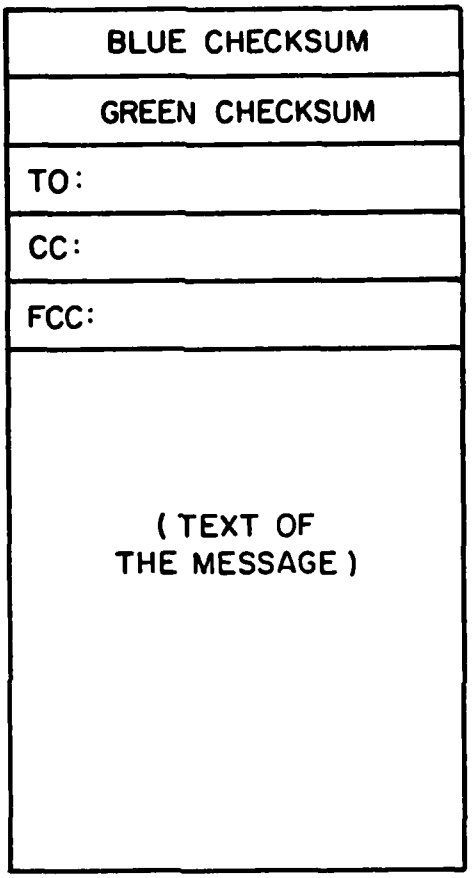


Figure 1. A "Safe Message"

in the message as an addressee. Only if both of these conditions are met is he given the message. It should be noted that this discussion applies not just to one user's receiving a new message from another, but to any access to data stored in the system. For example, if a user wants to examine a message he received earlier, perhaps to include it in a message he is composing, the system retrieves the message from the file system and then performs the check just described. The effect of this care is that there can be no security exposure possible from any code that moves Safe Messages within the message switch, so that this part of the code may be unverified and need not be executed twice. If some error is about to permit a message addressed to A to be delivered to B, the check just mentioned will prevent the delivery. On the other hand, if a hardware error changes the addressee from C to D, the checksum will fail. (Of course, some errors may keep the system from working at all, but security will not be compromised.)

An important idea behind this scheme is that it is extremely unlikely that "normal" code could fail in such a way as to generate a valid checksum. This assumption is clearly not justified in the presence of malicious programming and so would not be acceptable in an operating system environment in which users could program the computer. It makes good sense in the present situation, however, because checksumming algorithms are sufficiently specialized that it is unlikely in the extreme that anything could "accidently" produce a correct checksum. This is particularly true of CRC checksums.

2.2.2 Software to Protect Against Hardware Faults

The basic approach to insulating the system from the effects of hardware errors is to perform twice, using completely independent hardware, those computations in which a hardware failure could permit an exposure. Our intention is to structure the program so as to minimize the amount of program that must be treated this way. These sections are the critical parts of the program, the non-critical parts being those sections in which hardware errors can do no harm. An important first step is to identify the critical and non-critical parts. For example, calculating a checksum for a Safe Message is critical, because we must be sure that the message contains only what it is supposed to contain; while moving a Safe Message through the system is non-critical, for the reasons already mentioned.

It is important that the blue and green paths involve completely distinct hardware, with different processors using different copies of the code, different memories for data, and different paths between the resources. If I/O is done it must be done over different I/O busses. All of this is necessary to insure that no systematic hardware error or alteration in the program can have the same effect on both blue and green. It then makes sense to speak of blue and green hardware, such as blue processors, green busses, green memory, etc. A particular piece of hardware need not be forever one color -- it can change under program control providing that we insure that it never is involved with both halves of a computation.

Consider a join where blue and green computations meet to become orange. We carry the computations separately up to the point where each of blue and green has computed a checksum. We

use differing checksum methods for blue and green, perhaps different coefficients in a CRC polynomial. We now attach both checksums to either copy of the message, and call that copy orange. Note that, at least in principle, it is not necessary at this point to check that blue and green have produced identical messages. If they have not, the error will be detected the first time the orange message is delivered to blue and green code; since one of them will detect an incorrect checksum, the message will be discarded. In practice it may be advantageous to check for identical text at the join to detect the error sooner rather than later, on the grounds that error recovery is then easier.

An example of a join occurs at the completion of message composition. A conceptually simple strategy for a join at that point is as follows. The message is collected separately by blue and green code, each receiving the typed characters over a separate I/O path. When the entire message is composed, blue and green each computes its own checksum. Both checksums are attached to one copy (say, the blue one), and that copy of the message is labeled orange and dispatched to its destinations. Although this method works, any error in either the blue or the green copy will not be detected until the orange message is sent to blue and green code later, when one of them will detect an incorrect checksum. This may not be until the message is received at its destination, a place perhaps remote in distance and time. At that point the system can only discard the message without a trace. It cannot even report back to the originator of the message that it was not delivered, since the system has no safe way to know his identity. (It appears in the message, but no part of the message can be trusted.) If the blue and green copies are compared at the join, a discrepancy will still require

that the user retype the entire message, but at least he is warned, and at a time when he can still retype it. Such hardware errors should be rare occurrences. Nonetheless, a tradeoff must be made between efficiency and ease of recovery.

Consider now a point where we enter critical code. Here the orange execution path splits into blue and green paths. Each side copies the data into its own private work area (or one side could use the orange copy to save time), and then each checks its own color checksum. It is necessary to be sure that each side has a message that is identical to what was created originally. This must be the case if both checksums are correct. It is important that green and blue have independent copies of the data before checking the checksum, in case the common copy is altered after the checksums are verified. It is also important that the code (or hardware) that verifies a checksum be different from the code (or hardware) that calculates it, or an error in this code (or hardware) could go undetected and permit leaks.

An example of such a fork occurs after a message has been received and is being prepared for delivery. The received data, a Safe Message as described previously, is orange. Green and blue code each make a copy, and each then checks its own checksum. They then check in with each other, and if both checksums are correct they proceed with the delivery.

2.2.3 Hardware to Protect Against Software Bugs

To protect the message switch from security violations caused by errors in the software, we follow a five step process: (1) We partition the code into critical and non-critical parts. The former consists of those parts of the program for which correct execution is essential to prevent security leaks, and

the latter consists of those parts in which incorrect code might cause system failure but cannot cause security leaks. (2) The critical parts of the code are executed twice, on different hardware each time, to detect any hardware error that might compromise security. (3) We use standard verification techniques to insure that there are no bugs in the critical parts of the program. (4) We provide special hardware to insulate the system from bad effects that might result from bugs in unverified code. (5) We provide a barrier in hardware and software (using the other four techniques) which permits only Safe Messages to leave the system.

The part of the system that executes unprotected is very much like a security kernel that is often used in secure operating systems. We do not in this Quarterly Technical Report discuss the software problems involved in implementing such a security kernel. Instead, we address a more basic topic: Pluribus's lack of the necessary hardware protection mechanisms to provide an isolated environment for the unverified portion of the program. As we shall see, it is relatively easy to provide this hardware for the Pluribus. The remainder of the section assumes detailed knowledge of Pluribus architecture.

The Pluribus processor does not have a traditional protection mechanism, with distinct Executive and User modes. A new card called a "Protect Card" must be developed to provide these mechanisms. This card would sit on each processor bus and watch the activities of the two processors on that bus. The card would be able to tell the difference between references created by the two processors by examining the key bits associated with each reference. Rules already stated guarantee that the same protect card cannot be involved with both the green and the blue half of a critical computation, since it is improper for the same processor bus to be used for both.

The protect card keeps track of whether each processor is in Exec or User mode. A processor in Exec mode is subject to no restrictions; all accesses are allowed to continue. In User mode, however, access is only allowed to a portion of the address space. A good scheme is to specify for each 4K window of processor address space a lower limit for User space.

A violation is then a read or write while in User mode to an address not in User space. When the protect card detects a violation, it causes the violating processor to receive a Quit (non-existent memory trap). On a write violation the card will prevent the write from accessing memory. On a read, the data will be forced to all ones so that the processor will not be able to see the data. The data in memory will of course be unchanged.

On any interrupt or Quit, including that caused by a violation, Exec mode will be entered. Presumably the interrupt vectors will be part of Exec space and thus the location to which control is transferred in Exec space can be trusted. To leave Exec mode, a particular address which is recognized by the protection card is written. Unfortunately, this cannot be done from Exec space, since a violation would then occur on the subsequent instruction fetch. Thus the "leave Exec" command and appropriate return instructions must be copied into User space before each use (to trust them) and then jumped to.

Status reporting logic is also required in the protect card to report the current processor mode and to indicate the cause and address of the most recent Quit.

A call on the security kernel can now be implemented by just referencing a non-existent location. Parameters can be left anywhere in memory, since when in Exec mode the processor can access

anything. I/O devices can all be excluded from User space so that all I/O must be done through the kernel. The protect card would add a small overhead to every reference on the processor bus, probably on the order of 15%.

An optional feature which we might also implement on this card would be a watchdog timer to force a return to Exec mode if the processor spends more than a specific amount of time in User mode. Normally the program would be sure to "check in" periodically by reentering the Exec to hold off this timer. Although this feature does not explicitly add to the security of the system, it would improve reliability.

The second hardware change to the Pluribus to provide a secure system is to establish a barrier against improperly exhibiting any message or message format. This is a two step process:

1. Verified software being executed in Blue and Green hardware assures that the output port corresponds to a valid addressee of the message.

2. The hardware at the output port assures that the message was acceptable to Blue and Green.

The first step is reasonably straightforward. The second step, however, could be implemented in several ways. For example, each line to a terminal could be served by two independent interfaces on output. One would be assigned to and operated by blue, the other by green. The outputs of these two interfaces would be compared exterior to the computer, and only if they agree would the output be passed on to the user. This requires doubling all of the hardware, and requires some buffering and synchronization.

Another approach would be to have the blue and green software place the output port number in the message and include it in the checksum. Orange code could then set up the interface -- the interface would check the port number and checksum before passing the message to the user. This approach requires buffering for each line within the interface for the line so that the entire message (or piece of the message) can be held in the interface until the checksum can be verified. This leads to problems on echoing input.

Several other approaches like these are also under consideration and must be evaluated before the appropriate one for this environment can be selected.

2.3 Sizing

Part of our task is to determine the hardware configuration required to operate a message switch on Pluribus. To start with, so as to give us some constraints within which to work, we have made some arbitrary decisions. We are considering a system designed to support 2000 users. The system will be a host on a network which, for convenience, we are assuming to be similar to the ARPANET. About half of the users will be on terminals connected directly to the system (i.e., local users), and the other half will access the system via the network. See also the discussion in Section 2.4.3.

2.3.1 A Model for the Message Switch

In order to determine the hardware configuration needed, it is necessary to know something about what the system does. We early decided that specification of a user interface was not a necessary part of this project, since BBN already has much experience in the area with other projects. Instead, we have decided to assume that the system to be built will do no more computation than is done by MAILSYS on TENEX. Thus, we must determine the computational load placed on TENEX by MAILSYS; then we can apply appropriate multipliers to determine loading on Pluribus. Note that this is a conservative approach to sizing, since it depends on a mail system (MAILSYS on TENEX) known to be very complex. A simpler mail system such as MSG would impose smaller demands. We have chosen to study MAILSYS both because it seems prudent to be conservative and because it is convenient to study. (All the needed expertise is local.)

Our plan is to model a "typical" user of MAILSYS, and then to use measurement tools to be built into MAILSYS to determine numeric values for the parameters of use. At any moment, our typical user is in one of five modes:

- Read — A reader is merely reading new mail, looking at one message after another at reading speed.
- Process mail — This is disposing of a message in some way, such as writing it on a file, deleting it, etc. This does not include composing a reply.
- Write — This is composing a message to be sent out. It includes the various addressing tasks that create the header, as well as composing the the text of the message.
- Scan old mail — This involves looking at old files and doing things with them.
- Think — "What should I do next?"

The common mode of use is alternating Read and Process, with an occasional Write interspersed. Scan is relatively uncommon.

To size the system then, we first determine the percentage of time spent in each of these modes, and we then determine the computational load each mode places on the system. Both of these tasks can be done relatively easily once suitable instrumentation has been attached to MAILSYS. Experiments have been outlined that will supply us with the required sizing parameters, but they are as yet incomplete.

2.3.2 Some Preliminary Data

Some data on system loading has been gathered by distributing a questionnaire to selected users of Hermes, the current experimental version of MAILSYS. These data were derived from 13 returns.

The average session in Hermes lasted 1.3 clock hours, using 33.6 seconds of CPU time. An average of 1.4 messages was sent. The average message took 5.6 clock minutes to compose and send, for an expenditure of 5.17 CPU seconds. These times do not include MAILER, the program that actually sends messages to their destination. When Hermes, or any other message system on TENEX, has a message to send, it places a copy of it in the user's directory as a file with a name like [--UNSENT-MAIL--].SMITH to indicate a message for SMITH. A separate program called MAILER looks in all directories from time to time for such file names and mails them. The time and resources used by MAILER will ultimately have to be determined and included in our statistics.

Startup time in entering Hermes averaged 8.68 seconds. (Improvements already under way will reduce this time.) There are no further details yet available on how the rest of the time was spent. It seems clear though that these users spent much of that 1.3 hours doing things not related to Hermes. (This corresponds to "think" time in our model.)

Clearly we need much more data to be able to make a meaningful sizing estimate. One interesting fact does stand out: Since a user using 33.6 CPU seconds in 1.3 hours is using

about 0.7% of TENEX's computing resource, it appears that the maximum capacity of TENEX is about 140 MAILSYS users. However, this figure is undoubtedly too high, both because of queuing effects and peak to average considerations. Queuing will degrade operation if the system runs above about 80% of capacity over relatively short periods of time. We do not know what the peak to average ratio is for a message system but 2 to 1 should not be suprising. Thus a TENEX could on the average only support about 60 users.

A different approach has been some analytic studies of two tasks in Hermes. If the number of characters in the message is N, the time taken to print it is about

$$0.7 + 0.001*N$$

CPU seconds. The elapsed time is, of course, dependent on the typing rate of the console.

The SEND subcommand takes a fully composed message and places copies of it in the user's directory for MAILER to send out later. If the number of addressees is A, and the number of characters in the message is N, then SEND uses about

$$1.2 + 0.25*A + 0.000013*N*A$$

CPU seconds to do its work. Here "addressees" include file copies as well as all destinations in fields such as "To:", "Cc:", etc.

2.4 Other Considerations

Our efforts on and current understanding of 1) security mechanisms for the message switch and 2) sizing approaches to its implementation were discussed in the previous sections. Many other issues have been under consideration during the first half of this contract; our initial ideas on some of these are described later in this section. However, the issues in the list below are less well resolved and will be the subject of our efforts during the next quarter. The duplication of some listed items and sections of this report should be taken to mean that we feel that our ideas are still in a rough form.

1. How does the message switch interface to the network? A path seems necessary which would permit the transmission of secure messages across the network to both secure hosts and secure users. This problem requires encryption and may interact with on-going PLI efforts.

2. How can we provide secure and unsecure ports into the message switch? It seems now that if we make all ports secure, we can protect against leaks to unsecure users. If the switch is connected to a network, it must be able to talk to unsecure hosts and users (without encryption) as well as to secure users.

3. What savings are possible by addressing a specific security environment — for example, one in which all of the users and the switch are in a secure environment with no unclassified users? As another example, what is saved by eliminating security entirely? Is there still a savings if the ability to upgrade to a full security system must be retained?

4. How does the swapping mechanism work? What mass storage devices should be used? If the program and buffer requirements are too large, how should mass storage be used for temporary storage? How are "Safe Things" stored in the swapping and file systems?

5. How will the disk or other mass storage device be interfaced to the Pluribus? Does an interface for an acceptable disk exist or must one be constructed?

6. What is the major program structure? What is the schematic of data flow in the system? A preliminary version of this has been developed.

7. What is the hardware configuration for a prototype system? How does it expand? What is the relationship between Pluribus size, switch capacity, and switch cost?

The remainder of this section describes our thoughts on implementation language and the message switch interfaces to the network and the user.

2.4.1 High Order Language

It has been our thought that the software for the Pluribus-based message switch should be written in a high order language (HOL). In addition to the usual reasons for using a HOL (ease of programming, improved readability, improved maintainability, etc.), there are two reasons specific to this application.

1. Since several existing mail systems (Hermes and MSG) are written in a HOL, there is the possibility of using existing code rather than having to recode from scratch.

2. Software verification is probably feasible only for programs written in a HOL.

Given these advantages for using a HOL, why is there any question about using one? Two objections stand out: existence of a compiler, and efficiency of the code. These turn out to be closely related. There currently exists no compiler for any HOL that produces code for the SUE, the processor used in Pluribus. Thus, a decision to use a HOL for the message switch would impose the necessity to create a compiler. Efficiency of the compiler code then becomes an issue. The message switch will be a high bandwidth processor, requiring truly efficient code. All existing Pluribus coding has been written in assembly language. A compiler that produces really high quality code at the state of the art in compiler production is an expensive item. Of course, use of a less efficient compiler can be compensated for in a Pluribus environment by using more hardware, so the issues involved in the tradeoff are relatively straightforward.

In spite of these objections, the possible payoffs appear to be large enough that we have been investigating the matter further. A compiler for SUE would not run on Pluribus but would be a cross-compiler from another machine —— such as TENEX. Since the SUE processor is similar in many ways to the PDP-11, we have investigated the possibility of modifying some existing compiler for the PDP-11 that runs on TENEX. Two obvious choices are BCPL and BLISS.

Two compilers for BCPL currently run on TENEX —— one for the PDP-10 and one for the PDP-11. The former has just gone

through an extensive improvement process and now produces moderately good code. Although one might guess that a compiler for SUE could be most readily produced by modifying the PDP-11 compiler, careful study has revealed that the PDP-10 compiler is a better starting point. In addition to the fact that the PDP-10 compiler is a better compiler, there is the problem that the PDP-11 compiler makes extensive use of the memory-to-memory opcodes in the PDP-11 —— opcodes that are not present in the SUE. We estimate that in perhaps six person months we could modify the existing TENEX BCPL compiler for the PDP-10 so that it would produce moderately good SUE code.

The BLISS language was developed at Carnegie Mellon University as a system programming language for the PDP-10; BLIS11 is a BLISS compiler for the PDP-11 that runs on the PDP-10 and can be run under TENEX. The people at Carnegie have put a lot of effort into code optimization, so that BLIS11 produces very high quality code —— probably about as good as the average programmer, although a skilled programmer does better when he is trying hard. The code generation strategies used in BLIS11 turn out to be more easily adapted to SUE than those used in the BCPL compiler, so we estimate that in about three person months we might have a really high quality BLISS compiler for the SUE, running on TENEX.

The verification problems to be solved suggest PASCAL as a candidate, since ARPA appears to be going in the direction of a derivative of PASCAL for its security work. This is hard for us to evaluate, since we presently lack details of the derivative under consideration. In general, PASCAL compilers require a run-time support package, which would be at best awkward in the Pluribus environment. (Neither BCPL nor BLIS11 requires such a package.)

Also, the PASCAL compiler for the PDP-10 does not produce particularly efficient code.

No decision has been made as yet in this area. PASCAL has too many unknowns for us to evaluate it properly. The compiler situation for BLISS11 is much better than that for BCPL, although BCPL has the advantage that all the needed expertise is in house. As a language, we prefer BCPL to BLISS, although the differences are not of great significance. BLISS has no "goto" statement, sometimes a moderately serious lack. Further, one could not easily be added, since the code optimization strategies which make the compiler so attractive are heavily dependent on there being no "goto".

Ignoring temporarily the issue of which HOL to use, we have given some thought to what modifications might be needed in any HOL to adapt it to the somewhat unusual environment presented by Pluribus. For convenience, our thinking has been in terms of BCPL. We note the following problems that would have to be solved as part of an effort to produce a useful BCPL compiler for Pluribus, running on TENEX:

1. We must modify an existing compiler to produce SUE code.

2. The Pluribus uses map registers to permit a processor with a 16-bit address to access 2^{20} bytes of memory. Programmers find that dealing with map registers efficiently is a significant part of writing good Pluribus code. Ideally, the compiler would totally insulate the programmer from this task, but this might be too much to hope for. The proper answer is probably to provide the programmer with linguistic

constructs suitable for advising the compiler about how best to handle map registers.

3. Life in a multi-processor environment requires certain special linguistic constructs, such as those to deal with interlocks, explicit parallel processing, etc.

4. Pluribus code is broken into short (in time) segments called strips, and in each Pluribus application there is the requirement that no strip execute for more than some number of milliseconds. The compiler may be able to help the programmer deal with strips, but it will probably be necessary for the programmer to advise.

5. Although there is no "operating system" as such on Pluribus, there is the reliability package that is part of all Pluribus systems. This is written in assembly language, and provisions will be needed to interface the HOL to it.

6. BCPL, BLISS and PASCAL require a stack at run time. It is not immediately obvious what this means in a multi-processor. Should there be a stack per processor? Or one per Pluribus PID level? Or a stack per user?

2.4.2 Discovery of Parallelism

In addition to its capability of supporting a reliable and secure message switch, Pluribus also offers the capability of high throughput through application of its multiple processors. Furthermore, as has been mentioned above, the task of message switching is in many ways a "natural" for a multi-processor, due to the parallelism inherent in the message-switching function.

This brings us to an important issue, however: to what extent do we expect parallelism to be discovered by the compiler, and to what extent do we expect the programmer to specify it? Our view is that the compiler will do very little sophisticated parallelism discovery. Automatic discovery of parallelism is a highly complex task which should not be placed in series with development of a message switch. Instead, we expect that the programmer will maintain control over the parallel processing in the algorithm. We mention the issue of parallelism discovery to present our views about what is within the scope of this project.

2.4.3 Terminal Access and Network Attachment

It is important to decide how user terminals will be interfaced to the message switch, and how the message switch will be interfaced to the network. We believe that the message switch should ultimately support several different (optional) methods of terminal access and network attachment, but the most basic configuration should be the message switch as simply a network host with all terminal access from the network.

Given this configuration, a number of modes of use of the message switch are possible:

1. With no additional features, the switch can support users at terminals throughout the network. Of course, this requires a standard set of protocols for terminal access to the message switch. A very natural set of protocols to choose would be the normal ARPANET host/host and TELNET protocols.

2. Local terminal access could be supported by the addition of the Pluribus TIP software package to the message switch. In this case, the Pluribus IMP code should also be added to provide the packet switching function between the message switching application software and the terminal handling TIP software, both taking the form of software-hosts in the Pluribus IMP computer. This configuration would be sufficient for a stand-alone configuration of the message switch and its terminals, not connected to any network. This same message switch configuration could also be connected to a network with the whole system appearing to the network to be a single host.

3. Next, one can consider the possibility of the Pluribus IMP resident in the message switch actually being a part of a network, a configuration which might be useful in some cases.

4. Assuming some sort of network connection, it would be possible for the message switch to take advantage of various network resources, such as a large archival store.

In summary, the basic configuration is simple a network host. To this basic configuration any of the other above-mentioned configurations are relatively simple additions of existing software packages, perhaps in modified form.

2.4.4 User Interface

Earlier in this report, we mentioned that we were not concerned with what user interface (i.e., message system) would be chosen for the message switch, but that BBN's Hermes would be studied as one example. A few additional words of explanation

are in order. ARPA is currently sponsoring research to determine the proper form for a message system. As we are not part of this effort, we feel that no good purpose would be served by our lobbying for yet another specific message system.

We do have a few comments, however:

1. We propose that only a single message system be implemented in the message switch.

2. We propose that the chosen message system communicate with other message systems using a standard message protocol, leaving open the possibility of replacing the selected message system by another.

3. While we do not propose to invent still another message system, it is possible that the existing message system chosen to be implemented will have to be modified somewhat to facilitate implementation on the Pluribus. It would be good if those people considering message system design and research keep in mind the possibility of a Pluribus implementation.

4. We are drawn towards selection of one of the simpler message systems being proposed rather than one of the more elaborate ones.

2.4.5 Bulk Storage

To meet the message storage requirements of the message switch there must be some form of bulk memory. This bulk memory serves two purposes: short term memory (including swapping) and

archival storage (including "file copies"). We are investigating several mass storage technologies for this application including conventional disc units and electron beam memory devices. Conventional disc units have the advantage of being a well-known technology. Electron beam memory devices, on the other hand, offer advantages such as low access time, while being relatively unproven in real applications. While a local memory device is clearly necessary for swapping, archival storage could be provided through the network on some other network host.

REFERENCES

1. F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings 42, June 1973, pp. 529-537; also in Advances in Computer Communications, W.W. Chu (ed.), Artech House Inc., 1974, pp. 329-337; also in Computer Communication Networks, R.L. Grimsdale and F.F. Kuo (eds.), Proceedings of the NATO Advanced Study Institute of September 1973, Sussex, England, published by Noordhoff International Publishing, Leyden, The Netherlands, 1975, pp. 159-180; also in Computer Communications, P.E. Green and R.W. Lucky (eds.), IEEE Press, 1975, pp. 366-374.

2. "Pluribus Document 2: System Handbook," BBN Report No. 2930, January 1975.

3. S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Heart, "Pluribus -- A Reliable Multiprocessor," AFIPS Conference Proceedings 44, May 1975, pp. 551-559.

4. W.B. Barker, "A Multiprocessor Design," BBN Report No. 3126, September 1975.

5. W.F. Mann, S.M. Ornstein, and M.F. Kraley, "A Network-Oriented Multiprocessor Front-End Handling Many Hosts and Hundreds of Terminals", to be presented at the 1976 National Computer Conference, New York, New York, June 1976.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A103409
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Interface Message Processors for the ARPA Network ~~QPR No.~~ 5	5. TYPE OF REPORT & PERIOD COVERED 1/1/76 - 31/3/76
	6. PERFORMING ORG. REPORT NUMBER BBN-3276
7. AUTHOR(s) F. Heart	8. CONTRACT OR GRANT NUMBER(s) F08606-75-C-0032
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2351 Program Element Codes 62301E,62706E,62708E
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE April 1976
	13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Range Measurements Laboratory Building 981 Patrick Air Force Base Florida 32925	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Discribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computers and Communication, Store and Forward Communication, ARPANET, Packets, Packet-switching, message switching, Interface Message Processor, IMP, Pluribus

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This quarterly Technical Report provides an interim report on the suitability of the Pluribus computer to provide a message handling service of large capacity and high reliability while meeting stringent security requirements.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PROJECT: A-2357-IT

NO. CYS RECD: 3

DATE RECD: 5/10/76

ROUTING: 1. MIS/KOETHER

2. _____

3. _____